Computer Programming 4 (Object Oriented Prog)

Clarification and Notes

1. Rule number 1

Before you write any code:

Make sure that you know what you have to do.

You cannot successfully write any program if you do not know what it is suppsoed to do.

2. In the real world .

you will frequently be asked to write a program but it will not be clear what you must do.

It is up to you to discuss with the client/customer and to work out exactly what the program is supposed to do.

In this case, Use Cases are very good at specifying what happens.

USE CASES

=========

Even if you are not asked for them, do up a use case to show that you are clear about what is going on.

E.g.

A customer is at Heuston station, and wants to go to Galway leaving at 10am tomorrow. Wants a return and a student ticket.

At that point, you might notice a few things.

1. An ATM in the station will know where the customer is starting from, but an online service, does not know where the customer is starting from. So, you need to clarify this.

2. A customer usually chooses a route first (i.e. source and destination), then they choose a time, then they choose the type of ticket.

Remember as well, that the system will know how many seats are available and let the customer know if the particular journey is full. There is no point choosing a student ticket, if the train you want to travel on is full, so choose the train and time before choosing the type of ticket.

Another use case would be a manager who wants to view the total seats occupied for a particular journey and also the total amount of money that has been spent on tickets on a particular journey.

Again, good diagrams can help you discuss the problem with a client who may not know much about what a program can do.

At this stage, we have thought about the problem but still haven't decided what we are going to do.

Let's make it very simple. Let's assume that there we will only sell tickets from Dublin to Galway and that there are no intermediate stops. The train has 100 seats. There is one train a day. And there are Adult/Child/Student tickets. There are single and return tickets.

Now, at last we know what we have to do and it is a very simplified system. But that is fine. Get a simple system to work first and then consider more complicated versions.

One final point. The Use Case should encourage you to think of a real example of how the system will be used. When you buy a ticket, you would not be asked your age. You would be asked whether you would want an Adult or a Child ticket. So, do not create a Customer class with an age attribute.

Inheritance

===========

It might not be clear how to use inheritance.

1. You can only directly inherit from one thing. (In C++ you do have multiple inheritance, but that feature doesn't exist in any sane language.)

2. Inheritance should make sense. It makes sense that a Cat is a type of Animal, or that a Car is a type of Vehicle, or that a Circle is a type of Shape. It does not make sense that a Customer is a type of ticket.

Inheritance is a way of saying that some item is a type of something else. Or more accurately, a subtype.

Polymorphism

============

It may be possible to use polymorphism in the ticket type.

There are many types of ticket. E.g. adult/child/student, first class, second class, single, day return, monthly return, commuter ticket.

Some students recognise a difference between the type of person using the ticket (adult/child/student) and fact that a ticket can be single or return.

You can't handle it all with subtypes, unless you wanted to create a subtype for an adult with single ticket and another for adult with return. That would not be sensible and creates more work than is needed. We will look at this problem a little later in the module.

For the moment, we do what we always do when faced with a difficulty. Just simply ignore return tickets.

For this assignment, just assume that the only ticket types are adult/child/student. You might wonder why I am saying that these are ticket types when really they are different types of people. Of course, it is a person who is a student. A ticket isn't a student and a ticket doesn't go to university. But, for this application, it makes sense to say that the ticket is a special type of ticket which can be described by the customer who pays for it.

You could also use polymorphism for the customer, but for the purposes of this application, the customer doesn't really do very much except pay for the ticket and occupy a seat and the way that they pay for the ticket and occupy the seat doesn't change depending on what subtype they are, so for the purposes of this assignment, customer polymorphism is not useful.

If you are using polymorphism, you need to think of what the base class will do. Customers collect tickets at an ATM. They use them to get past the ticket gates and present them for inspection if required. What operations should a ticket perform.

If there are a limited number of tickets, then how do you manage that?

What classes should your system have?

======================================

One approach to coming up with classes is to read a description of the problem and consider making each noun a class.

Finally, remember that a well designed system will have each class responsible for its own task and will communicate clearly with the other classes. Classes should not communicate more than necessary with other classes as this is likely to make it harder to modify any individual class in isolation.

Last modified: Monday, 15 March 2021, 12:40 PM

◄ UML Assignment: Train ticketing System (part 2)

Jump to...

Refactoring Example ►